

General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.
- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.
- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.
- This document is paginated as submitted by the original source.
- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Report No. 76-0006
Contract No. NAS8-31222

(NASA-CR-144191) MOSS, AN EVALUATION OF
SOFTWARE ENGINEERING TECHNIQUES Final
Report (M&S Computing, Inc.) 40 p HC \$4.00
CSCI 09B

N76-18822

G3/61 18485
Unclas

FINAL REPORT

MOSS AN EVALUATION OF SOFTWARE ENGINEERING TECHNIQUES

January 30, 1976

Prepared for:

George C. Marshall Space Flight Center
NASA
Marshall Space Flight Center, Alabama 35812

M&S COMPUTING, INC.

PREFACE

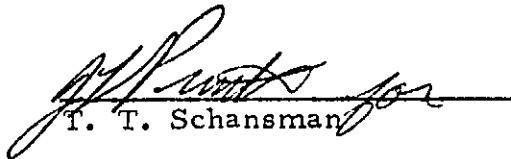
This document contains an evaluation of the software engineering techniques used for the development of MOSS, Modular Operating System for SUMC. MOSS is a general purpose real time operating system which was being developed for MSFC's Concept Verification Test (CVT) program under Contract No. NAS8-31222.

Each of the software engineering techniques is described and evaluated based on the experience of the MOSS project. Recommendations for the use of these techniques on future MSFC software projects are also given.

Prepared by:

J. R. Bounds
J. L. Pruitt

Approved by:


T. T. Schansman

MAR 1970
RECEIVED
NASA STI FACILITY
INPUT BRANCH

TABLE OF CONTENTS

<u>Section</u>	<u>Page</u>
LIST OF FIGURES	iii
1. INTRODUCTION	1
1.1 Chief Programmer Team Summary	2
1.2 Top-Down Development Summary	2
1.3 Structured Programming	2
1.4 HOL Utilization Summary	2
1.5 Virtual Memory Summary	3
1.6 Concurrent Hardware/Software Development Summary	3
2. SOFTWARE DEVELOPMENT TECHNIQUES	5
2.1 Chief Programmer Team	5
2.1.1 Chief Programmer Team Principles	5
2.1.2 MOSS Chief Programmer Team	6
2.1.3 Chief Programmer Team Evaluation	9
2.2 Top-Down Development	11
2.2.1 Top-Down Development Principles	11
2.2.2 Hierarchical Modularity Principles	12
2.2.3 MOSS Top-Down Development	16
2.2.4 Top-Down Development Evaluation	17
2.3 Structured Programming	17
2.3.1 Structured Programming Principles	17
2.3.2 MOSS Structured Programming	19
2.3.3 Structured Programming Evaluation	19
2.4 MOSS Top-Level Design	19
2.4.1 MOSS Layer Specifications	21
2.4.2 MOSS Processes	26

TABLE OF CONTENTS
(Continued)

<u>Section</u>	<u>Page</u>
3. HIGH ORDER LANGUAGE UTILIZATION	29
3.1 SUE Language Capabilities	30
3.2 SUE Evaluation	31
4. VIRTUAL MEMORY IN REAL TIME OPERATING SYSTEMS	33
4.1 Virtual Memory Impact on User Applications	33
4.2 Virtual Memory Impact on MOSS	34
5. CONCURRENT HARDWARE/SOFTWARE DEVELOPMENT	35
5.1 Concurrent Hardware/Software Development Evaluation	35

LIST OF FIGURES

<u>No.</u>	<u>Title</u>	<u>Page</u>
2-1	Chief Programmer Team Comparison	7
2-2	Staged Implementation	10
2-3	Intrafunction Modularity	14
2-4	Standard Simple Program Structure Flow Charts	20
2-5	MOSS Structure Overview	22

1. INTRODUCTION

M&S Computing has been engaged in the design and implementation of MOSS, a general purpose, real time operating system for the SUMC computer, over a period of several years. The major objective of the project has been to produce a reliable, modifiable, and understandable operating system. To meet these objectives, the performance of this work has been committed to the utilization of state-of-the-art software engineering techniques.

In recent years, considerable effort has been devoted to the development of large and complex operating systems. These systems provide services such as on-line file structures, interactive capabilities, and sophisticated resource management facilities that were not available on earlier systems. Aerospace computers, as well as commercial computers, are undergoing these changes to more sophisticated and complex capabilities due to advances in circuit and packaging technology. Their operating systems also are showing increased complexity with the addition of services and capabilities formerly reserved for commercial systems.

Unfortunately, this complexity has brought with it a number of serious problems. The cost of building such systems is significant. Development time is long and unpredictable, system modification is difficult, and the software is never completely debugged. These problems are a result of the fact that the systems are intellectually unmanageable; i.e., they are so complex that it is impossible for an individual to understand the operation of all of the components of the system and how they interact.

To overcome these difficulties, new software engineering techniques have been proposed in workshops and technical papers. These techniques have been applied widely for both general purpose and real time systems with general success. The use of these advanced techniques for the implementation of a large, general purpose, real time operating system has, however, not been attempted before the MOSS project. This project presents an opportunity to perform a detailed evaluation of these software engineering techniques as applied to the MOSS implementation. Such an evaluation would provide valuable experience for future MSFC software production endeavors. The proper use of software engineering techniques can make a significant contribution toward the goal of low cost, reliable software for space systems.

This document contains the results of the evaluation and assessment of the software engineering techniques used in the design and implementation of MOSS. Section 2 contains a review and assessment of the software development techniques utilized on MOSS. Section 3 contains a discussion of the utilization of a High Order Language (HOL) for operating system implementation. In particular, the use of SUE for MOSS implementation is reviewed. Section 4 deals

with the assessment of a virtual memory environment in a real time operating system. Then, Section 5 discusses the concurrent development of hardware and software.

The remainder of this section summarizes the conclusions and recommendations of the evaluation.

1.1 Chief Programmer Team Summary

Conventional chief programmer team principles were found to be an unresponsive organization for developing large, complex operating systems. An adapted chief programmer team, giving more responsibility to senior team members, was found to be a better organization.

The major problem was, however, the overall size and complexity of operating systems. This problem can be dealt with by using staged implementation - developing the system in smaller, manageable parts (called stages).

1.2 Top-Down Development Summary

Top-down development principles were found to focus too exclusively on the functional structure of the system, ignoring the data structure and flow. Operating systems are especially dependent on the data structure and flow.

A technique, such as composite design, which emphasizes the data aspects of development was recommended for future operating system projects. Concurrent attention should be given to both the functional structure and data structure of the system.

1.3 Structured Programming

Structured programming was found to aid the general clarity of the code produced in the MOSS project. It was found, however, that structured programming does not lend itself to handling exception cases.

1.4 HOL Utilization Summary

High Order Language use for operating system development was generally found to be useful. Specific problems with the SUE System Language were noted. It was recommended that selection criteria for an HOL include adequate documentation and a proven compiler in addition to language capabilities suited to operating systems development.

1.5 Virtual Memory Summary

Virtual memory was found to be a positive factor contributing to a better design of MOSS. It was, however, found to be a negative factor in the development of user applications requiring in-depth knowledge of virtual memory policies and concepts.

1.6 Concurrent Hardware/Software Development Summary

Concurrent hardware/software development was found to be very beneficial in providing evaluation and feedback to both disciplines. It was recommended that formal review procedures be established to resolve conflicts in designs. It was further recommended that software design precede hardware design somewhat so that potential conflicts may be resolved before the hardware is fixed.

2. SOFTWARE DEVELOPMENT TECHNIQUES

Three design and implementation principles were used in the MOSS project. They are:

- o Chief Programmer Teams,
- o Top-Down Development,
- o Structured Programming.

In this section each of these techniques is evaluated. First, the basic principles of each technique are discussed, followed by the adaptations made for the MOSS project. Each technique is then evaluated in terms of its effectiveness and productivity and recommendations for use in future development efforts.

2.1 Chief Programmer Team

Chief Programmer Team is a management discipline applied to the organization of personnel for software development. The methodology is directed toward reducing software costs and improving software quality by changing from the traditional loosely structured team of programmers to a more structured team of programming specialists who work under defined operational disciplines.

2.1.1 Chief Programmer Team Principles

The Chief Programmer Team is characterized by defining the positions and responsibilities of each member of the team. The team is typically composed of the following positions:

- o Chief Programmer - The chief programmer is a senior level programmer and analyst who is responsible for the development of the programming system in all respects. This person carries a technical responsibility for the project, including higher echelon coordination. He produces the critical core of the programming system in detailed code himself, directly specifies all other code required for system implementation, and reviews and oversees integration of that code.
- o Backup Programmer - The backup programmer is a senior level programmer and analyst who functions in full support of the chief programmer at a detailed task

level so that he is constantly in position to assume the chief programmer's responsibility temporarily or permanently. He may be called upon to explore alternative design approaches, independent test planning, or other special tasks but serves normally as an active participant in technical design, internal supervision, and external management functions.

- o Librarian - The librarian is a programmer technician or secretary. He will assemble, compile, linkage-edit, and test-run programs as requested by project programmers. The librarian has direct responsibility for task of maintaining the development support library.
- o Team Members - The team is a flexible module that can be supplemented with additional programmers, analysts, or technicians commensurate with the workscope. As the design and development workscope evolve, either additional programmers can be added to a given team to write the programs specified by the chief programmer, or components of the overall design can flow to other teams for more detailed design and coding.

2.1.2 MOSS Chief Programmer Team

As previously stated, software development techniques such as the Chief Programmer Team provide a methodology for development of software. Each application will require modifications so that the techniques are adapted to the environment of the specific task.

The design phase of MOSS utilized multiple chief programmer teams. Each team assumed design responsibility for several major areas of the system. The design progressed fairly well within the team principles given above. The major problem encountered was a lack of communication and design integration across the various teams. The lack of interteam effort hindered the overall design progress. As the design of the individual parts of the system were nearing completion, mass meetings of all teams were required to work out the interfacing problems of the separately designed areas. These meetings were very unproductive use of the project personnel and produced only workable, not conceptually clean, design compromises to tie the overall design together.

The implementation phase of the project employed an adapted Chief Programmer Team concept as illustrated by Figure 2-1. The adaptations were aimed at eliminating the lack of interteam communication experienced during the design phase. The chief programmer retained most of the responsibilities

CHIEF PROGRAMMER TEAM COMPARISON

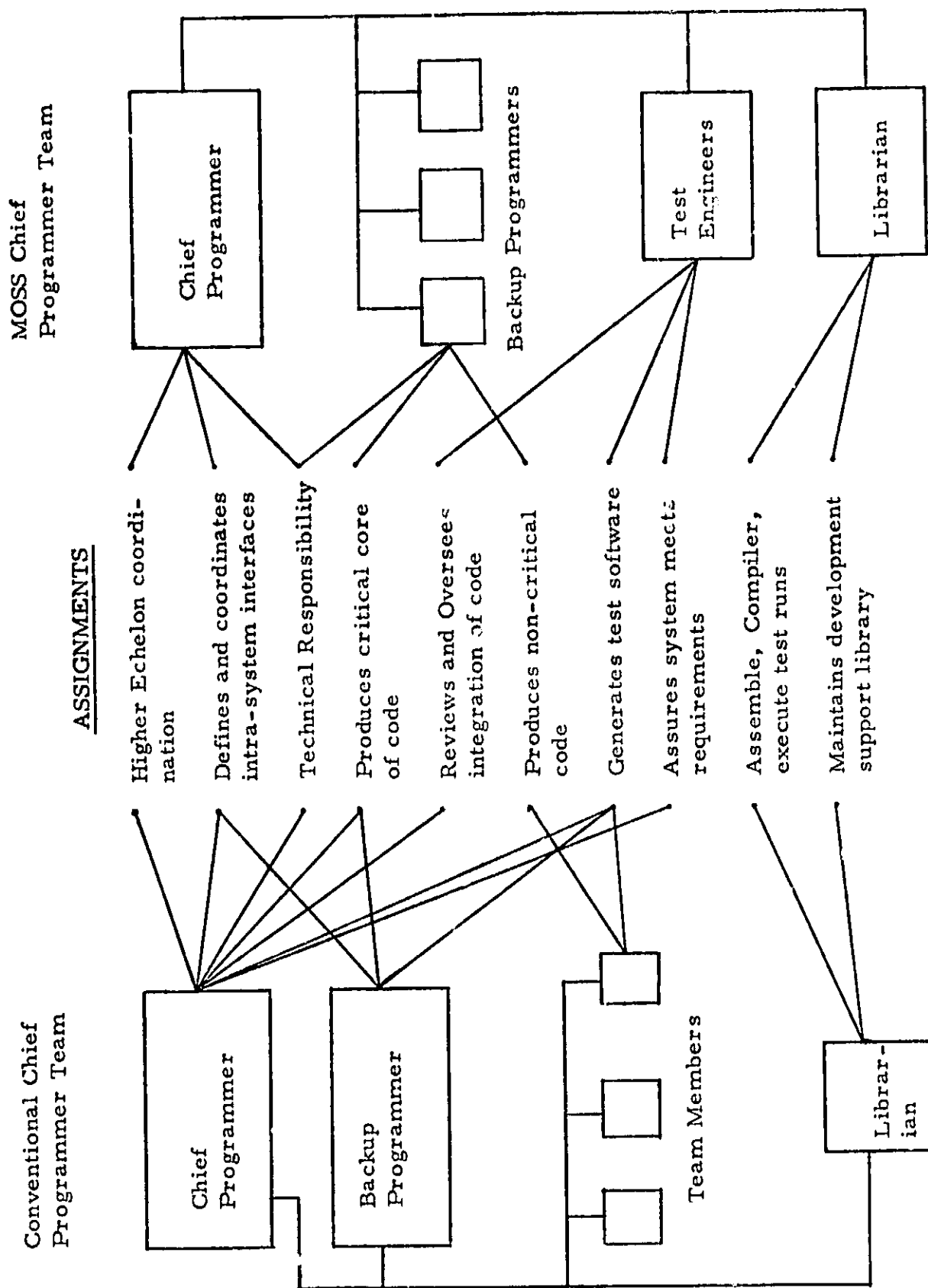


Figure 2-1

identified for that position. However, the chief programmer did not produce the critical code of the system. As discussed in a following section, MOSS was developed using an architecture based on hierarchical levels of system functions. Because of this architecture, the definition and identification of critical code was difficult. Each level of the system contained critical code to the overall system functions. Individual team members were given responsibility for a level of the system and produced the code for that level.

The chief programmer had the specific responsibility for defining and coordinating the interfaces between levels. The overall integrity of the design with respect to the requirements and specifications remained a direct responsibility of the chief programmer. He was also responsible for the identification and coding of the common system data structures.

Due to the size and overall complexity of MOSS, all of the team members were senior level programmers. Each team member acted as a backup programmer, assuming primary responsibility for the technical correctness of several major levels of the system. This organization required each team member to maintain a functional knowledge of the overall system and a detailed knowledge of how his level fit into the total design.

Each team member was in a position to be utilized to explore alternative design approaches, independent test planning, or other special tasks. The assignment of these type tasks was generally keyed to the area of responsibility of the team member. For example, the team member responsible for the Process Management function would be used to perform a special assignment related to process switching hardware.

A significant refinement to the chief programmer team was instituted during the implementation phase. The position of test engineer was added to the team. The test engineer was responsible for generating all support software for testing and evaluation during top-down program development. The test engineer must have knowledge of both the system requirements and the design meeting those requirements. This position was established because of the need for an integrated approach to the system testing activities. Many of the testing activities for the individual levels of the system required simulations (test stubs) of other levels. The test engineer was responsible for this coordination. He provided the overall continuity to the system checkout the same as the chief programmer provides continuity to system design and implementation.

2.1.3 Chief Programmer Team Evaluation

The use of the chief programmer team on the MOSS project met with many problems and did not, in general, provide a responsive organization for dealing with the recurring problems of developing a large, complex operating system. The problem of lack of interteam communication previously noted in the design phase were partially resolved by the adapted approach established for the implementation phase. Even though the design of MOSS was intended to divide the system into discrete, manageable parts, the interaction among the parts remained complex. This caused problems for the team members dealing with the detailed knowledge of their level of the system as well as the details of other levels necessary to understand the complex interactions.

The test engineering position has proved to be a valuable refinement to the chief programmer team concepts. The separation of the testing responsibility has encouraged the thorough checkout of programs and reduced temptation to shortchange program and system checkout for the sake of coding progress.

In general, the lack of success with the chief programmer team on the MOSS project can be attributed more to the size and complexity of MOSS than the chief programmer team organization. In fact, the adapted team principles, with the chief programmer responsible for the interaction of the parts of the system, have been responsible for maintaining control of the MOSS project.

Team organizations with well-defined position and responsibilities seem to provide a degree of control and responsiveness to implementation of large systems. Size and complexity, however, remain the determining criteria for the successful implementation of any programming system. The MOSS project was one of such size and complexity that successful completion on schedule would have been risky to predict.

Future NASA endeavors of this type should be based on a staged implementation. The system should be designed to be implemented in stages, each stage representing a complete system of manageable size and complexity. The overall system design should be completed at a high level of detail. Then, the detailed design and implementation for the first stage should be completed. The experience with the design gained from this implementation can then be fed back into the overall design to produce the necessary adjustments to and/or redesign of the system. Successive stages build on the earlier stages and implement further capabilities (see Figure 2-2). Each stage, thus implemented, is a working useful system. This approach not only controls size and complexity in small increments, but also aids management control of and visibility into the scheduled progress of the system development.

STAGED IMPLEMENTATION

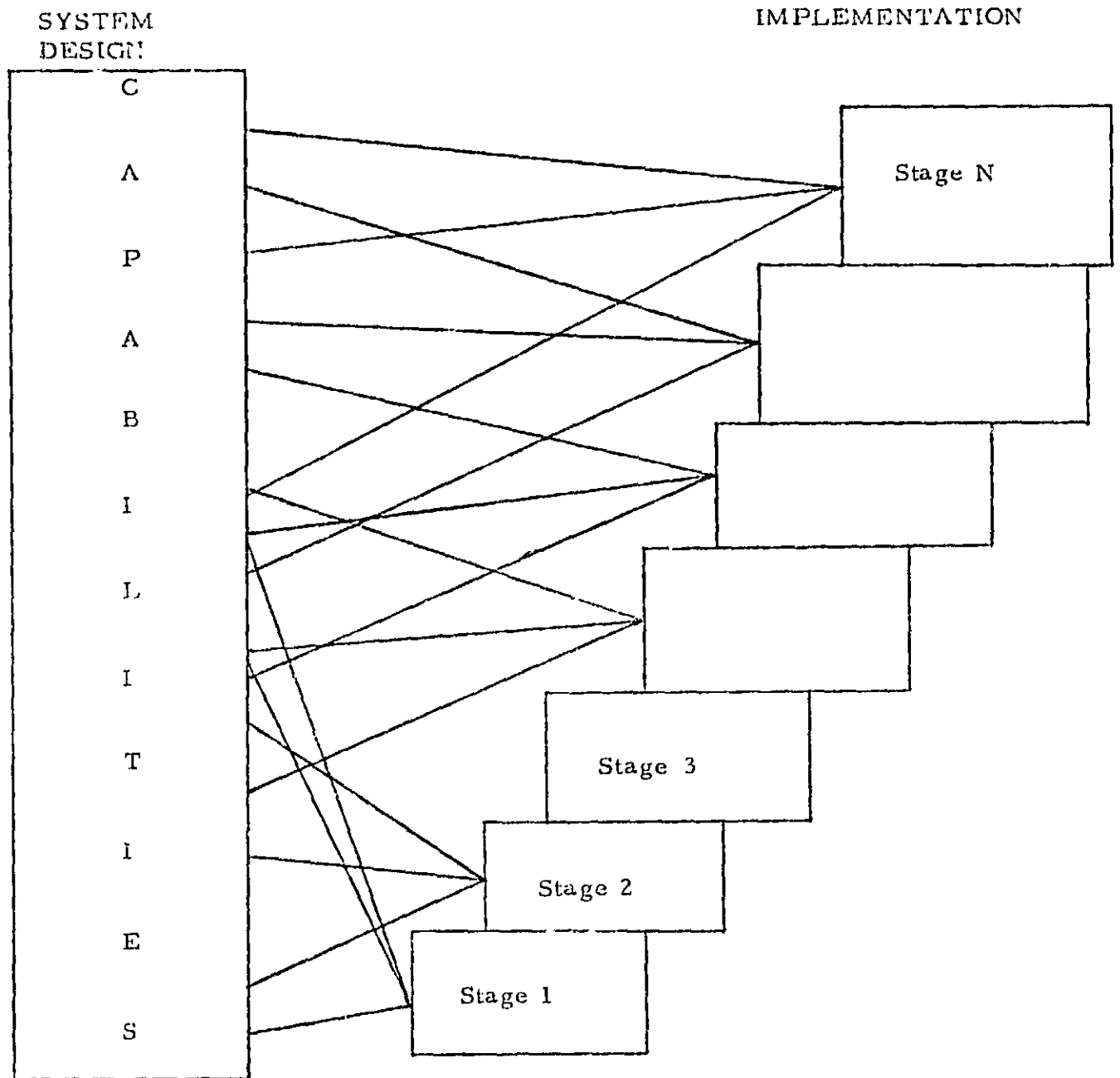


Figure 2-2

2.2 Top-Down Development

Top-down development is patterned after the approach to system design that requires programming to proceed from developing the control architecture (interface) statements and initial data definitions downward to developing and integrating the functional units.

Top-down development on the MOSS project encompassed hierarchical modularity techniques in addition to strict top-down development techniques. The principles for both of these techniques are described in this section.

2.2.1 Top-Down Development Principles

Top-down program design and implementation are an application of deductive analysis to programming. As a principle of programming, it requires that each procedure or module be capable of complete explication at some level of abstraction as a single block of code.

Each such block has a single entry at the top and a single exit at the bottom, and all external references must be to other such blocks at a lower level of abstraction (i. e., which can be represented within the main block by a functional description of the operation or operations performed). Such other blocks are only exceptionally terminal; i. e., return is always made to the main block upon normal completion unless the block involved is an exception handling routine.

A major stylistic characteristic which this principle is meant to enforce is that each block of code represents a complete function (whether in-line or out-of-line) at some level of abstraction and can be read literally from top to bottom typographically, without external references, unless detail at a lower level of abstraction is desired. It is important to recognize that the freedom to define levels of abstraction, to use lower level structures as elements of higher level structures, and to reference such lower level structures out-of-line can be used to defeat the intent of good structure and top-down design and implementation. Hence, it is absolutely essential that each block represent the accomplishment of a relatively simply defined function (at the next higher level of abstraction) and not an arbitrary collection of functions which cannot support a common abstract functional description. Further, every block must be used only as an element in a simple structure.

Top-down development requires that programming proceed from developing the control architectures (interface) statements and initial data definitions downward to developing and integrating the functional units. Top-down development permits programming to proceed naturally in parallel with the continual integration of system parts as they are programmed. With

top-down development, the highest level blocks of a system or subsystem are coded and tested first. Testing and integration starts with the highest level system blocks as soon as they are coded. Since these blocks will normally invoke or include lower level blocks, some code must exist for the next lower level blocks. This code, called a program stub, may immediately return control, may output a message for debugging purposes each time it is executed, or may provide a minimal subset of the functions required. These stubs are later expanded into fully functional blocks of code, which in turn, will require lower level blocks. Integration is, therefore, a continuous activity throughout the development process. During testing, the system executes the blocks that have been completed and uses stubs where they have not been completed. The developing system itself can support testing because the code that will interface with the newly added blocks has been previously integrated and tested and can be used to feed test data to the new segments. In fact, no major changes need be made to the test data as the system evolves, since it is always a "complete" system (from the point of view of the input that is being tested).

This approach also provides a basis for capturing performance data during the development cycle. By replacing each stub with a timing loop that utilizes the estimated run time for that function, the developing system becomes a model. As dummy routines are replaced with working code, the performance results can be appraised against the performance objectives. In a similar manner, storage allocation can be modeled.

Thus, it is possible to exercise and check the processing paths and system architecture at the higher level before initiating implementation of the lower levels. The important point is that program units at each level are fully integrated and tested as a composite system before coding of the next lower level is required. Thus, completion of coding occurs in phase with completion of the initial phase of integration.

Top-down development provides the ability to evolve the product in a manner that maintains the characteristic of always being operable, extremely modular, and always available for the successive levels of testing that accompany the corresponding levels of implementation. The quality of a system produced by using the top-down approach is increased through earlier detection and elimination of major design problems and coding errors. Top-down development also eliminates the need for writing drivers for unit testing and generates a system whose most complex blocks are also the most tested blocks.

2.2.2 Hierarchical Modularity Principles

A module is defined to be a block of code which may be independently compiled and which may, at least potentially, be independently loaded.

However, each module is constructed to implement one or more of a set of closely related functions at some analytical level. Two complementary principles of modularization (within a function and across all the functions in the system) were employed.

By application of the first principle, every module represents a block of code. The converse need not be true, since a block of code detailing an in-line structure will not be a module. At the lowest level, therefore, such a module is a sequence of simple structures or in-line nests of structures of statements. At the highest level, such a module would be a sequence of simple structures whose elements consist of in-line nests of structures, calls to lower level modules (i. e., out-of-line sequences of simple or nested structures), and statements in the programming language.

The highest level module describes the steps of a major function, the lowest level module (which does not contain calls to other modules) describes the steps of an atomic function, and the intermediate level modules describes the steps of functions of intermediate levels of complexity. That is, the intermediate functions are compounded of atomic functions, lower level intermediate functions, and statements in the programming language organized according to the rules of good structure. Figure 2-3 shows the breakdown of a function into intermediate and atomic level modules.

By application of the second principle of modularization, the entire set of hardware and software capabilities is segregated into a hierarchy of layers of capabilities, where the lowest layer in the hierarchy provides only the most elementary capabilities and each succeeding layer adds increasingly abstract and more general capabilities (e. g., memory management, I/O management, event management, etc.). Each layer of this hierarchy defines a virtual machine, and communication among different layers is closely restricted to permit the verification of each layer as an independent entity. This stratification of the complete system into a series of increasingly abstract machines permits verification of the whole as a collection of relatively simple (albeit increasingly abstract) machines rather than as a single entity of uncontrolled complexity. Each layer consists of a group of related components and conforms to the following rules:

- o Each layer owns certain resources to which other layers are not permitted direct access.
- o No layer requires any services of layers above it in the hierarchy.

INTRAFUNCTION MODULARITY

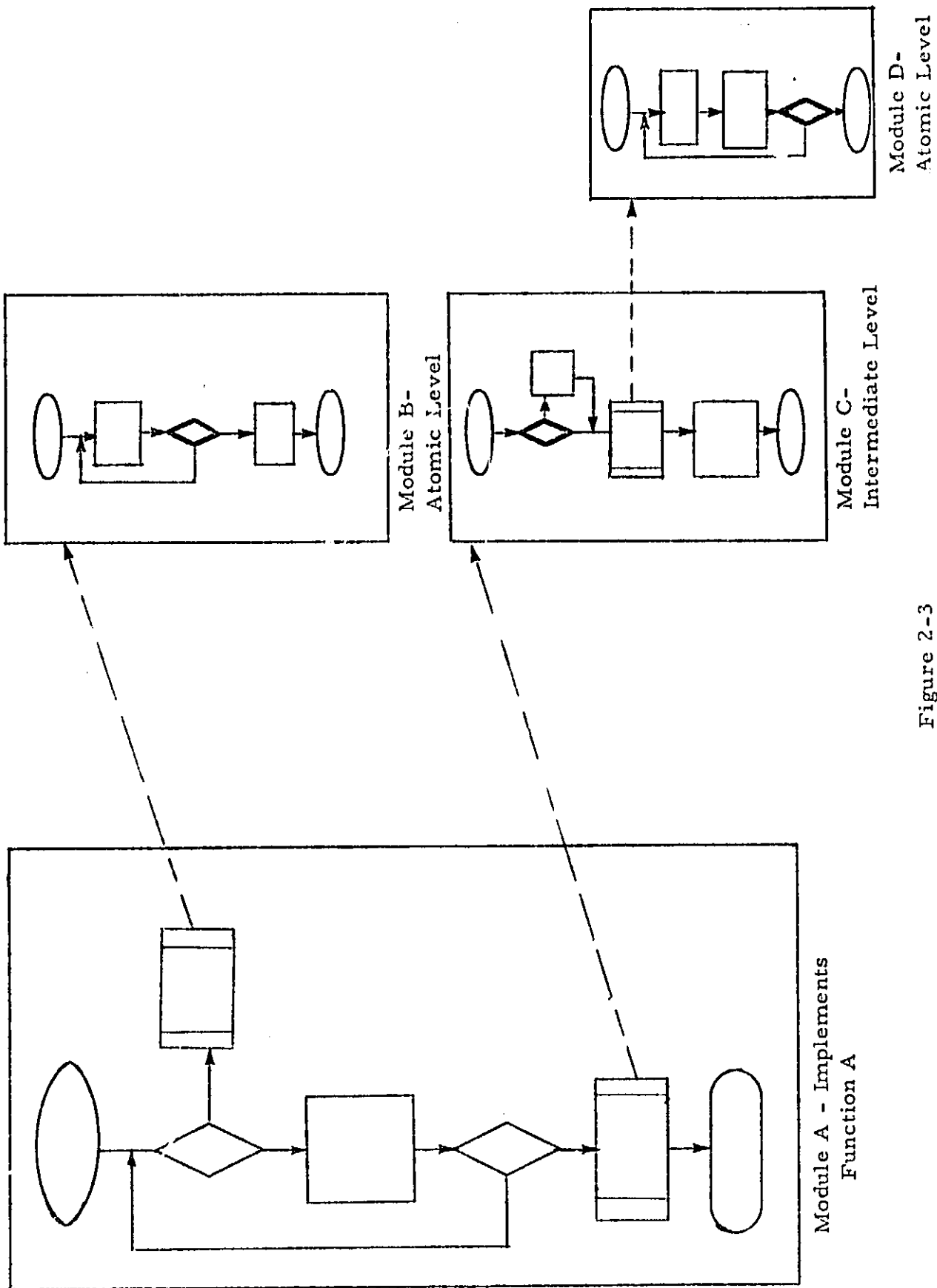


Figure 2-3

The components within a layer are classified as either external or internal. External components are capable of being invoked from higher levels to perform a requested service or to supply information obtained from its resources (resources owned by the layer in which the external component resides). Internal components perform functions required by the other components of the layer but are not capable of being invoked directly from any other layer.

The first step in the program design process is to select the layers. Each layer is selected to support or accomplish one or more of the following:

- o It maps virtual system capabilities or resources into less abstract virtual system capabilities or resources, or into physical system capabilities or resources.
- o It maps logical data structures into less abstract logical data structures or into physical data structures.
- o It simplifies the system by restricting the access of the system and the components of a layer to resources.
- o It provides services and/or manages resources for upper layers.

As each layer is defined, the functions performed by it and the resources it owns are specified. The interfaces among the layers and components are also established.

The next step in the design process consists of the evaluation and refinement of the layer structure and interfaces. Each layer is examined with respect to its relationship with the rest of the system. In particular, it is verified that the layers obey the previously stated layering rules and perform all of the functions required by the layers above them in the hierarchy. It is also verified that the overall design meets the system requirements. The verification process may point out inadequacies which are corrected by modifying existing layers or by establishing additional layers. When a satisfactory layer structure has been achieved, the process diagrams are completed.

As subsequent levels of detailed are specified, the internal components of the layers are defined. The following criteria are used:

- o A data structure or control block (e.g., the file directory is contained within a single component in the sense that the content and format of the data structure or control block are made available to that component only. That

component, therefore, controls all access and modification to information contained within the data structure or control block.

- o The sequence of operations required to perform a given well-defined function is contained within a single component.
- o Hardware dependent information is contained within a single component.

2.2.3 MOSS Top-Down Development

The development of MOSS took place in two relatively independent phases. The first phase was the design of the overall architecture of the system. During this phase, the basic parts (levels) of the system were defined and the interaction between them was established. The second phase of development covered the design and implementation of each level.

The MOSS architecture was based on hierarchical layers of partitions corresponding to levels of abstraction. A layer defines a level in the hierarchical structure of the system and contains one or more partitions whose interaction is limited and well understood. A partition is a group of functions which are related in their effects and share common resources (data structures and/or hardware features).

Within each partition there are internal and external functions. External functions may be invoked directly by functions of another partition. The external functions of a partition provide the primitive operations of the partition to other partitions in the system. Internal functions are used only within a partition and cannot be referenced from other partitions. The internal functions are derived from the decomposition of the partitions into modules which support information hiding.

Each partition was developed in a top-down manner from design through implementation. All phases of the development were expressed in the notation of the implementation language - SUE. Narrative features were developed for SUE to add the necessary descriptive information to the design. Each successive iteration of the design replaced more and more descriptive information with actual SUE code. The final iteration of the development produced the code for that partition.

2.2.4 Top-Down Development Evaluation

The use of top-down development lead to an understandable functional breakdown of the system at the top levels. This overall architecture provided a framework which remained stable as the design was modified to reflect changing requirements. New personnel of the project were able to understand the basic structure of MOSS and work within it. The top level design also successfully minimized the timing considerations which plague many real time system projects.

The successive refinements of the top-level design, however, lead to increasing complexity in the system. The significant problem was the poor development of the system data flow. The focus of development was too concerned with functional relationships of the system. The definition of data used by a function and the data to be passed among the functions did not fall out until late in the design/implementation process. This problem seems to be a shortcoming of top-down development techniques, rather than misuse by the MOSS team.

Operating Systems are more dependent upon and sensitive to the data organization and flow through the system than other types of programs. This is due to the many asynchronous activities (each with its own particular data flow) active in the system. Special attention should be given to the data definition and data flow in operating systems developments beyond that normally encouraged by top-down development principles. Techniques such as composite design which support the concurrent development of both the functional and data structures of the system should be used. At each step of the process, the two structures are compared and inconsistencies resolved.

2.3 Structured Programming

Programs coded in a traditional fashion usually use many branching instructions to handle the control logic of the program. Experience has now shown, however, that programs can be coded in a structured manner which greatly enhances code readability and maintainability. Structured programming is based on the principle that any program can be expressed in terms of basic logic structures, as discussed below.

2.3.1 Structure Programming Principles

Good program structure is defined to be one which can be shown to be a structure, albeit of finitely many compounded simple structures. A simple structure is defined to have one of the following forms;

Sequential

The sequential program structure is an ordered set of operations which are to be performed in order by some machine. It should be noted that the operands may themselves represent structures of less complex operands and hence the machine which shall perform them may be some arbitrarily abstract machine.

Conditional

The conditional program structure has as its simplest form: IF a THEN b ELSE c (where b is executed if a is true, and c is executed if a is not true). This can be expanded to the more general form, "CASE i," where only the element identified with case i and none of the elements identified with other cases is executed if i is true.

Iteration

The iteration program structure has as its simplest form: DO a WHILE b. Upon entry to this program structure, b is tested and, if true, a is executed and the program structure is reentered. If (or when) b is not true, this program structure is exited. Hence, a is executed for as many times as b is found to be true prior to each potential execution of a. This can be represented by the conditional: IF b THEN a' is the recursive procedure:

```
BEGIN PROCEDURE a'
    a
    IF b THEN a' ELSE exit
END PROCEDURE a'
```

A variation of the iteration program structure is the DO a UNTIL b where b is tested at the end of each iteration and the structure is reentered only if b is true. In this structure, a is performed at least once, regardless of the value of b.

Each of these three simple structures is a proper program (defined as a program with a single entry and a single exit). Large and arbitrarily complex programs may be developed by appropriately nesting these structures as elements within themselves. The logic flow of such a program always proceeds from a single entry to a single exit without arbitrary branching however complex such a structure becomes.

An arbitrary branch is defined to be any branch which connects physically separated portions of a simple structure, but never to be a branch to another simple structure at some lower level of abstraction from which a return is eventually made.

These simple structures can be represented by corresponding flow charts (see Figure 2-4).

2.3.2 MOSS Structured Programming

Structured programming on the MOSS project was inherent in the use of the SUE language. SUE contained only structured control statements. Section 4 covers the use of SUE for MOSS development.

2.3.3 Structured Programming Evaluation

Structured programming was not used as a separate technique, but was combined in the use of the SUE language. Thus, an evaluation of structured programming is hard to separate from the evaluation of the SUE language. This section will cover comments relating to the control structures provided by the SUE language.

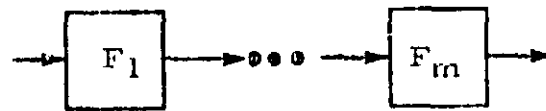
One general concern of structured code is that the principles do not generally lend themselves to handling the exception cases. Most examples of structured code in the literature used a call to a system routine to service an exception. For operating system development, however, there is no lower controlling code to call for exception handling. During design and implementation of these functions, the method usually turned into brute force coding. Research needs to be done to either expand the language or to gain further insight that would lead to acceptable solutions in this area.

2.4 MOSS Top-Level Design

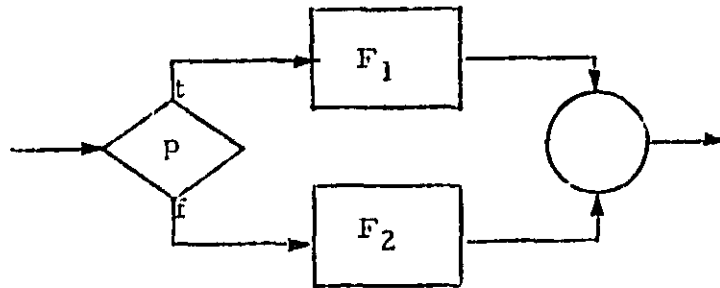
This section summarizes the top-level design of MOSS resulting from the application of the software development technique described previously.

The MOSS Operating System uses an architecture based on hierarchical levels of system functions overlaid dynamically by asynchronous cooperating processes carrying out the system activities. A major architectural decision was the separation of the static and dynamic structures of the system. The static structures, consisting of a hierarchy of functions, defines the basic framework of the subsystems and their interaction. The dynamic structure, consisting of cooperating processes representing user applications and asynchronous system activities, is superposed on the static framework.

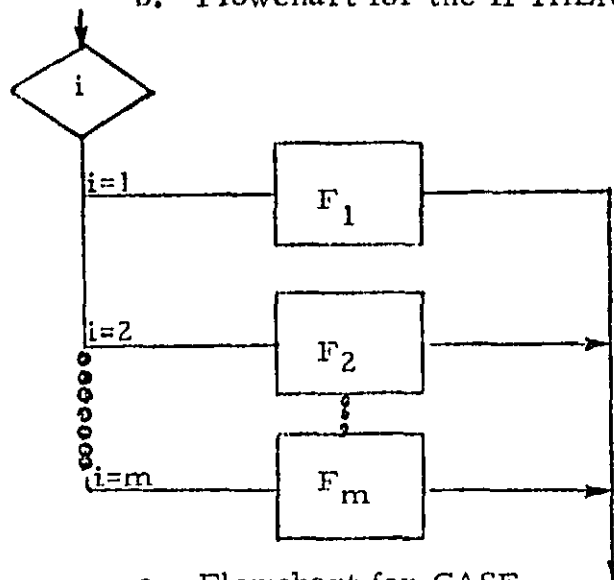
STANDARD SIMPLE PROGRAM STRUCTURE FLOW CHARTS



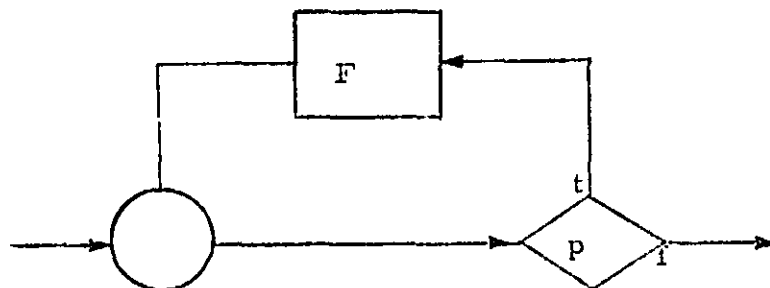
a. Flowchart for SEQUENTIAL structure



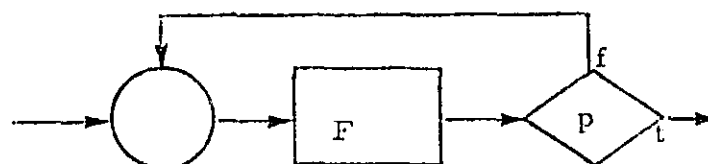
b. Flowchart for the IFTHENELSE program structure



c. Flowchart for CASE structure



d. Flowchart for DOWHILE program structure



e. Flowchart for DOUNTIL program structure

The architecture of MOSS has proven to be successful in achieving the basic design goal of modifiability. The hierarchical relationship of the MOSS partitions has remained unchanged since the early phases of the design while the dynamic structure and partition interfaces have been augmented to satisfy additional user requirements.

The concepts of exclusive ownership of resources by partitions and information hiding within modules has permitted the internal design of the partitions to be modified without affecting the interfaces into the partition or the design of other partitions.

The concepts of strictly controlling process communication has permitted processes to be modified and added without affecting other MOSS processes. For example, the data bus monitoring process has been added since the original design.

2.4.1 MOSS Layer Specifications

MOSS is organized into eleven hierarchical layers, each containing one or more partitions. This section describes each layer starting with the lowest hierarchical layer and proceeding to the highest (see Figure 2-5).

Layer 1

Layer 1 consists of the timer management, processor management, process management, exception handler, and log queue management partitions. These partitions provide basic system services which can be requested by all system partitions.

- o Timer Management Partition - This partition provides timing services for the MOSS processes. The partition controls the setting of an internal timer and the processing of the external interrupt generated when this timer expires (see Section 2.4.2).
- o Processor Management Partition - This partition maintains the status of the central processing unit and allocates the processor to the highest priority ready process. Time slicing of the processor is not utilized.
- o Process Management Partition - This partition controls the progress of processes in the MOSS environment. The partition creates processes, coordinates their activities, and deletes them. Process coordination is accomplished via the SINT and unconditional blocking/unblocking mechanisms.

MOSS STRUCTURE OVERVIEW

HARDWARE INTERRUPT LEVEL	LAYER	PARTITIONS
User	11	1 User tasks 2 Systems tasks
SVC	10	1 SVC handler 2 External control 3 Log management
	9	1 Sampling performance monitoring
	8	1 Program management
	7	1 Logical I/O 2 Console management
	6	1 I/O resource management
	5	1 Access management
	4	1 Event management
Page Trap	3	1 Memory management
I/O	2	1 Channel management
Timer Exceptions	1	1 Timer management 2 Processor management 3 Process management 4 Exception handler 5 Log queue management

Figure 2-5

The functions of the processor management and process management partitions produce a multi-programming environment.

- o Exception Handler Partition - This partition analyzes all detected errors in MOSS. Hardware/firmware detected errors are reported via the SUMC interrupt structure while software detected errors are reported via normal procedure invocation.
- o Log Queue Management Partition - This partition manages the in-core buffers of the system log. All entries into the system log are made via this partition. The partition activates the log management process whenever a log buffer becomes full in order to have the full log buffer written to the system log data set.

Layer 2

Layer 2 contains the channel management partition. This partition centralizes the control of the SUMC channel hardware. It is responsible for scheduling the channels, processing I/O interrupts, and maintaining the channels' logical status.

The partition is placed at this hierarchical level to enable the memory management partition to request paging I/O operations.

Layer 3

Layer 3 contains the memory management partition. This partition supports the abstraction of virtual memory; i.e., the ability to reference an address space which is relatively independent of physical memory and whose contents may actually be in main memory or in a backing store called external paging memory (EPM). This allows main memory to be shared among programs whose individual or total memory requirements exceed the main memory size.

Under MOSS each task is assigned a linear address space of 2^{24} bytes. Each address space is divided into four segments; task private, job common, system common, and MOSS private. There are not separate copies in memory of the last three segments for each task. The hardware permits the sharing of single copies of each job's common area, the system common area, and the MOSS private area.

Main memory is allocated on a job basis and varies between a user-defined minimum and optimum amount depending on the requirements of other jobs. Paging for all tasks of a job is done in the main memory allocated to the job. This scheme contributes to the repeatability of successive executions of a job; an important real time performance consideration.

When a page is referenced which is not in main memory, a page trap interrupt is generated. This internal interrupt is processed by the memory management partition which brings the required page into memory. However, the layering of MOSS prevents this interrupt from being honored if it is incurred by modules at or below the memory management partition. Therefore, such modules must always be locked in main memory.

Page swapping is performed on a page group basis, i.e., one to four pages of 1K bytes each. The SUMC virtual memory hardware supports address translation and page traps at the page group level.

Layer 4

Layer 4 contains the event management partition. This partition provides the services for communicating the occurrence of significant conditions (called events) between processes. The partition also provides the capability to suspend a process pending the occurrence of a logical combination of events.

Events do not exist in the MOSS environment until a process defines a condition to be an event. Although the event management partition maintains the status of events, it relies on other partitions to detect the conditions which constitute the event. That is, the reporting and processing of events has been centralized while event detection has been decentralized. This is consistent with the principal of dedicated ownership of resources discussed in Section 3.1.

Layer 5

Layer 5 contains the access management partition. This partition provides the services for controlling the access rights to selected resources in such a manner as to prevent deadlock situations between processes. A requesting process whose specified access request cannot be satisfied is blocked until the required resources become available.

Layer 6

Layer 6 contains the device management partition. This partition provides the services for allocating, scheduling, and maintaining the logical status of the SUMC I/O devices (except for the paging device controlled by the memory management partition).

The partition translates I/O requests into channel programs and queues the requests until the required device becomes available. The channel management partition is subsequently invoked to schedule the associated channel for execution of the channel program. The partition also performs I/O error recovery if required.

An actual situation involving this partition provides an example of the modifiability of MOSS: Because device characteristics and configurations are maintained by this partition and hidden from the other MOSS partitions, the other partitions could be designed and coded even though the device models and configurations were still undefined.

Layer 7

Layer 7 contains the logical I/O and console management partitions.

- o Logical I/O Partition - This partition provides the services for supporting I/O requests which are specified in terms of logical units rather than physical units. The partition performs two types of transformations; logical units are translated into one or more physical I/O functions. The device management partition is invoked to perform any required device I/O.
- o Console Management Partition - This partition coordinates the use of the system console among the MOSS processes. The partition supports the abstraction of a virtual console for each process. This enables a process to use its own virtual console without the necessity of coordinating input or output with other users.

Layer 8

Layer 8 contains the program management partition. This partition provides services for controlling the initiation, termination, and status of jobs and tasks. A job is defined as a unit of work which consists of one or more tasks. A task is the basic unit of work processed by MOSS and is the smallest entity competing for system resources.

Layer 9

Layer 9 contains the sample performance monitoring partition. This partition provides services for collecting system performance data at periodic time points.

The collected data indicates the current utilization of the system resources. This data is recorded and subsequently processed to provide a statistical report which indicates system bottlenecks and poorly utilized resources.

Layer 10

Layer 10 contains the SVC handler, external control, and log management partition.

- o SVC Handler Partition - This partition provides the interface between MOSS and tasks executing at the user level. All executing tasks must invoke this partition whenever MOSS services are required. The partition is invoked by an internal interrupt generated when a task executes a supervisor call (SVC) instruction.
- o External Control Partition - This partition provides services which permit the control of MOSS operation from an external source (e.g., the console typewriter). The partition contains functions which allow the system operator to obtain and/or modify the current status of the system.
- o Log Management Partition - This partition maintains the system log data set. Its primary purpose is to output the system log buffers which are filled by the log queue management partition.

Layer 11

Layer 11 contains the user task and system task partitions. These partitions may not execute privileged instructions or access the MOSS private segment of the address space.

- o User Task Partition - This partition contains all user programs.
- o System Task Partition - This partition provides various system support programs. System tasks are processed by MOSS in the same manner as user tasks. System tasks include linkers, loaders, reader/interpreters, and output writers.

2.4.2 MOSS Processes

The MOSS environment may be viewed as a set of cooperating processes where each process is a unit of work to which the processor may be allocated.

MOSS provides facilities for controlling the creation, execution, and deletion of individual processes, and also provides services for controlling communication and synchronization between processes.

In this section, each major MOSS process is identified and its purpose explained.

I/O Interrupt Handler Process

This process performs the initial interpretation of the data furnished with I/O interrupts. The process remains inactive until a hardware I/O interrupt signal is accepted by the processor.

If the I/O interrupt indicates that an active I/O request has completed, the interrupt data is passed to the process which requested the I/O and that process is reactivated. Such processes normally return to higher layer components which perform device-dependent analysis of the data. If the I/O interrupt is not associated with an active I/O request, the process activates the device attention process. In addition, if the I/O interrupt indicates that a channel or device has become available, the I/O interrupt handler process attempts to restart the channel with a request queued for that channel.

Timer Interval Expired Process

This process performs the functions associated with the expiration of a predefined time interval. These functions include the activation of any processes which have requested their own suspension until the expiration of the time interval and the resetting of the timer for the next time interval. This process is activated by the hardware interrupt generated when the processor's interval timer expires.

Device Attention Process

This process performs detailed interpretation of the data provided by I/O interrupts which are not associated with an active I/O request. This process is activated by the I/O interrupt handler process when an "unexpected" I/O interrupt is received.

Data Bus Monitoring Process

This process examines the status of selected real time data sensors (e.g., analog inputs) at periodic time points. The data obtained is used to maintain a representation of the devices in main memory. These data values may subsequently be used to satisfy requests to read the real time data. This method expedites the processing of requests for real time data and reduces the request load on the channels associated with the monitored devices.

The process is activated by the timer interval expired process at the beginning of each monitoring interval.

Job Process

MOSS creates a job process for each job. The job process performs the functions required for initiating and terminating a job.

Each job consists of one or more tasks. The job process creates and activates the task process for the initial task of a job. In addition, the job process is activated whenever a task of the job terminates. This permits the job process to control sequential scheduling of tasks and to determine when the job has completed.

Task Process

A task process is created for every task which executes within the MOSS environment. A task process is created when a request is received by MOSS to execute a task and exists until the task is terminated.

The task process is created and activated by the job process if the task is the initial task of the job or if the user requests sequential scheduling of the tasks of his job. Otherwise, task processes are created and activated in response to a task scheduling request from another task of the same job.

Sampling Performance Monitoring Process

This process collects basic system performance data and enters it into the system log. The process is activated by the timer interval expired process at each sampling time point.

External Control Process

This process carries out the commands which are entered at the system console. The process is activated by the device attention process when an external control command is read from the system console device.

Log Management Process

This process outputs a system log buffer to the system log data set. The process is activated by a log queue management function whenever a system log buffer becomes full. This function is a process because the writing of the log buffer is asynchronous to the other processes in the MOSS environment.

3. HIGH ORDER LANGUAGE UTILIZATION

Implementation of operating systems, especially real time operating systems, with an HOL has been avoided in the past. The major objections have concerned the excessive use of main memory and the lack of control over time critical areas. The experiences of other software projects where HOL was utilized have shown, however, greater productivity, maintainability, and readability for the complete system.

M&S Computing felt that the problems of using an HOL for an operating system were solvable and the apparent advantages outweighed the problems. The inclusion of virtual memory hardware in the SUMC reduced the problem of the size of the completed operating system. MOSS could be paged in a small amount of main memory. Timing problems in real time systems can rarely be isolated to inefficiencies in the code above. The design of the system also has a significant impact on system performance. The approach used for MOSS was to optimize the system design for maximum performance consistent with the clarity of the system. These design considerations were intended to limit the remaining timing problems to a small percentage of the code which could be identified from actual performance data. For those sections of code where timing was of critical nature, that code could be done in assembly language. Thus, a language with an in-line assembly language feature was desirable.

Selection of the proper HOL for implementation of the MOSS programs was restrained by several factors:

- o The language had to provide constructs that are in agreement with structured programming constructs.
- o The language had to reside in a S/360 and produce S/360 target code.
- o The language had to be commercially available and functional. (Time did not permit the development of an operating system HOL.)
- o A desirable feature would be in-line assembly language capabilities.
- o The language would serve as the design vehicle.

Several languages were considered for MOSS, including PL/S, HAL/S, and JOVIAL. The SUE language was a language developed by Project SUE at the University of Toronto as a HOL for operating systems.

SUE was chosen because the language capabilities offered the greatest possibilities for a structured programming environment. The SUE language provided the basic structured programming constructs.

3.1 SUE Language Capabilities

The SUE language is designed to facilitate separate compilation of hierarchically related programs. The structure of the language is intended to encourage the development of programs by conceptual refinement. Operations which are logically primitive at one level of abstraction are defined in terms of more basic operations at the next. Some levels of refinement are achieved by means of procedures, others by textual macros that are expanded at compile time.

The SUE system language has a rigid organization of procedures. Each procedure has a DATA block in which information is shared with its local (contained) procedures. This information includes the names of its parameters and returned values, and declarations of the local procedures and all common variables, types, and macros. CONTEXT blocks are similar to DATA blocks, but are restricted to constant information which is to be shared by the procedures of cooperating processes. Then each procedure has a program block which contains the executable code and purely local declarations.

Some of the features of SUE that are of significance in developing good code are discussed below.

- o SUE permits long identifiers with a pseudo-blank (underscore) to allow naming variables in a meaningful manner.
- o SUE has a program constant feature which permits the ability to define an identifier as equal to a constant.
- o SUE has a macro expansion feature that instructs the compiler to expand code keyed to the macro name.
- o SUE provided for automatic indentation of the SUE statements that produced highly readable code.

The SUE language was extended to accept and recognize narrative statements and expressions. This extension allowed the use of SUE for design notation as well as the actual system code. The MOSS design was expressed in this pseudo-code at all levels. The narrative information was replaced by code with the successive top-down refinements of the design.

3.2 SUE Evaluation

A total evaluation of the use of SUE on the MOSS project is impossible since the system was not implemented and fully tested. One of the major reasons for the use of an HOL was increased productivity. This objective cannot be evaluated at this point in the implementation except on a subjective level. The general feeling is the SUE did contribute to the production of MOSS code.

The implementation of SUE (the compiler) was a disappointment. A considerable amount of effort was required to correct bugs in the compiler. Also, the compiler syntax checking and recovery left a lot to be desired. An excessive number of computer runs were required to correct syntax diagnostics. Often, a computer run was required to process beyond each "severe" error detected by the compiler. Considering the state of development of the compiler, SUE was a poor choice for the MOSS project. The HOL compiler selected should have been one which was suited for production coding and had a proven record of previous use.

SUE was a difficult language to learn. There was continuing resistance and confusion about capabilities provided and their use. Aside from the nature of the language itself, many of these problems can be attributed to the lack of instructional and other supporting manuals. Also the diagnostic messages produced by the compiler failed to provide the necessary feedback to learn the language through "hands on" experience.

A negative aspect of the SUE language itself was the lack of data initialization features. The only way a data structure could be assigned initial values was to provide a module for that purpose. The lack of this capability caused an unnecessary increase in the amount of code which had to be generated for MOSS.

Even though considerable difficulties were encountered with the SUE language, it is recommended that HOL's be employed in future operating system implementations. The problems encountered with SUE were not due to its applicability to operating systems but rather with the compiler and the lack of supporting documentation. The criteria for HOL selection must go beyond the language capabilities suited to operating systems. The language must be easy to learn and accompanied by sufficient documentation. The compiler must have a proven record of previous use.

4. VIRTUAL MEMORY IN REAL TIME OPERATING SYSTEMS

Virtual memory has seldom been used in real time operating systems since response requirements normally conflict with the time delays incurred during paging I/O operations. If virtual memory was used, real time response was often achieved by locking the entire real time program in main memory, thus, wasting an important resource.

MOSS attempted to support virtual memory with policies which would allow real time application programs to utilize this important resource without sacrificing performance. To accomplish this goal, the following design decisions were made:

- o Each job would be allocated a user-specified range of main memory into which the tasks of his job would be paged.
- o A two-level address translation mechanism would be used to reduce paging rates while simultaneously reducing the memory fragmentation problem.
- o Only user-specified portions of tasks would be locked in main memory.

These decisions result in better utilization of the main memory resource since the user need only lock the time-critical sections of his program, permitting the remainder of the program to be paged in an efficient manner. In addition, allocating a user-specified range of main memory to the job contributes to run repeatability; an important real time performance consideration.

4.1 Virtual Memory Impact on User Applications

The MOSS virtual memory management policies require the user to be familiar with the basic concepts of the virtual memory policies and to cooperate with other users in the efficient use of the memory resources.

The user must specify both a minimum and an optimum amount of main memory to be allocated to his job. However, these are difficult parameters for the user to derive. Insufficient main memory will result in thrashing while excessive main memory will result in poor utilization of the resource and reduce unnecessarily the amount of main memory available to other users.

The user must also specify which modules of his program are to be locked in main memory. This policy requires that the user design the structure of his program in a manner which makes efficient use of the memory resource. That is, user programs should be structured such that the portions

which must be locked in main memory are isolated in "small" modules. This, again, is often a difficult requirement for the user to fulfill.

4.2 Virtual Memory Impact on MOSS

Allowing the user to control key parameters in the management of the memory resources results in a policy of trust between MOSS and the user. For example, although MOSS monitors user programs for thrashing caused by insufficient main memory, it does not detect requests for main memory which are excessive. Therefore, users may abuse the virtual memory services provided by MOSS. The services of MOSS permit the efficient use of virtual memory in a multitasking real time environment only if the users are knowledgeable, conscientious, and cooperate among themselves.

Memory management policies which rely on users to provide critical parameters should provide the user with data which will aid him in deriving these parameters. For example, paging rate profiles for a task would assist the user in determining whether more or less main memory should be allocated.

The hierarchical layering of the MOSS influenced the implementation of virtual memory. The rules of layering prohibit modules in layers below the memory management layer from utilizing the memory management functions. This implies that page fault interrupts cannot be incurred by such modules. Therefore, to prevent these page faults, all modules below the hierarchical layer of memory management must be permanently locked in main memory. Hence, some modules are locked in main memory only because of the hierarchical layering which could otherwise be paged. The higher the memory management functions are in the system hierarchy, the more main memory is required to hold the locked portion of the system which is lower in the hierarchy. Fortunately, in MOSS, the memory management functions are at a low hierarchical layer. Therefore, only a small portion of MOSS was required to be locked to satisfy the layering requirements.

Virtual memory influenced the decision to code MOSS in a High Order Language. Since the programmer has little control over what code a HOL compiler will produce, he has less control over the size of the program than would an assembly language programmer. However, virtual memory minimizes this argument against the use of High Order Languages since system size is no longer as critical. The design of MOSS showed both positive and negative aspects of virtual memory in real time systems. Virtual memory was a positive factor in the design of MOSS, contributing to the use of an HOL for the implementation and making memory limitations largely transparent to the system. On the other hand, user applications were required to be deeply involved with the virtual memory concepts and policies. The user was required to derive difficult performance parameters and structure his programs in accordance with them.

5. CONCURRENT HARDWARE/SOFTWARE DEVELOPMENT

One of the brighter spots of the MOSS project was that the value of concurrent hardware and software design was neatly demonstrated. Since the cost of software has passed the cost of hardware in virtually all applications, the need for this approach is a must.

In particular, the concurrent hardware/software development was visible in the communications concerning virtual memory, the interrupt structure, and the process switching mechanism. The interaction between the hardware and software designers was particularly productive in these areas. Software and hardware must be mated in a manner that extracts the best from both disciplines. Hardware that is not controllable or usable by software, represents an undersirable posture. On the other hand, software must be designed to fully utilize all hardware capabilities.

5.1 Concurrent Hardware/Software Development Evaluation

Although the overall interaction between the hardware and software development was beneficial, there was room for improvement in several areas. The first problem was that the interaction was never well-defined. Procedures should have been established for resolving conflicting hardware and software designs. It is recommended that the software and hardware disciplines participate in formal reviews. The software personnel should present their understanding of how the hardware operates and how the software will utilize it. This provides the means to identify any misunderstanding of the specification. If there is a problem, both sides are then in a better posture to discuss alternatives as they have an understanding of how the other discipline functions.

Another problem was that many of the potential conflicts between the software and hardware were not discovered until the lowest level details of the software design had been worked out. By this time, it was often too late to affect the hardware design. It is recommended that the software development precede the hardware development by a sufficient time frame to maintain the flexibility for hardware design decisions. This does not eliminate the need for close coordination of the software and hardware design. It allows the two efforts to be better paced for concurrent progress.